# Python Introduction and Math Operations

## 1 Purpose

- Execute individual *Python* commands
- Learn how to create and run simple *Python* scripts
- Learn about three important, commonly used object types
- Generate "printed" output on the screen
- Perform fundamental math operations
- Create and use variables (names)

## 2 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapters 1 and 2
- *The Coder's Apprentice*, Pieter Spronck, chapters 3 and 4
- *A Practical Introduction to Python Programming*, Brian Heinold, chapters 1, 3, 10, and 22
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapters 2 and 3

## 3 Executing *Python* Commands

*Python* commands can be executed without needing to create programs (actually scripts). Doing so allows *Python* to be used as a powerful calculator and to test chunks of code.

- From a *Python* interpreter command prompt
  - This environment is also known as the REPL (Read, Evaluate, Print, and Loop)
  - The prompt usually consists of three arrows; `>>>`
  - Typed commands are executed when the `[return]` key is pressed
- From a code block in a *Jupyter* notebook
  - Execute a single command by pressing `[shift]-[enter]` after typing it
  - Type multiple lines by pressing `[return]` after each line then press `[shift]-[enter]` to execute all of the lines

A *Python* prompt ready to enter a command

```
% python
Python 3.9.13 (main, Oct 13 2022, 16:12:19)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Entering commands/expressions at a *Python* prompt

```
>>> print("Hello")
Hello
>>> 2 + 3
5
>>> (3**2 + 4**2)**0.5
5.0
```

## 4 Editing and Running Scripts

### 4.1 Editing Scripts

- *Python* scripts must be written in plain text
    - Good - stand-alone text editing applications work well
    - Better - an editor built into a *Python* capable integrated development environment (IDE)
- Word processors like *Word* should never be used for writing or editing scripts
- *Python* installations have a built-in text editor/IDE named *IDLE*
- Dedicated coding text editors, like *Visual Studio Code* (*VS Code*), often add nice features; including the ability to run scripts directly from the editor
- Good, beginner-friendly IDEs
    - *Thonny* - can install a "local" version of *Python* from within the IDE
    - *Mu* - best for *CircuitPython*

### 4.2 Naming scripts

- No spaces in the name
- Must start with a letter or an underscore character
- May include numbers
- Must end with `.py` file extension
- Some valid script names…
    - `stress_calculator.py`
    - `fibonacci.py`
    - `_calculator.py`
    - `easy_as_123.py`
    - `uscs2si.py`

### 4.3 Running Scripts

- From a terminal or console command line prompt such as Windows Power Shell or the MacOS Terminal app
    - Type `python` or `python3` followed by the script name, including the `.py` extension
    - Examples…
        - `python hello.py`
        - `python3 hello.py`

- From a *Python* prompt
  - Change to the directory containing the script file before starting *Python*
  - Start *Python*, type `import script_name` at the `>>>` prompt, and press `[return]`

Running a script from a *Python* prompt

```
>>> import hello
Hello, World!
```

- From a *Jupyter* Notebook
  - Place the script file in same directory as the *Jupyter* notebook
  - Execute `run script.py` in a code cell using the actual script name, i.e. `run hello.py`

## 5 The Pythonic Method

There are programming methods and principles that are fairly unique to *Python*. Sometimes the use of these methods and principles is referred to as being **Pythonic**. The development of *Python* as a language has been guided by a number of principles. Executing `import this` provides a glimpse into the philosophy of the original *Python* developers.

Zen of *Python*

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The *Python* developers also have a bit of a sense of humor. They also desire to make *Python* code easy to use and read. Execute the following commands for a little bit of *Python* fun.

Some *Python* fun

```
>>> import __hello__
Hello world!

>>> import antigravity   # may not work from a Jupyter notebook

>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

Guido von Rossum, the founder and original author of *Python*, was given the title BDFL - Benevolent Dictator For Life - by the *Python* community (although he is now retired from the position). An April Fool's joke regarding the BDFL was published as PEP (Python Enhancement Proposal) 401 in 2009 (https://peps.python.org/pep-0401/) announcing that Guido's title was changed to Benevolent Dictator Emeritus Vacationing Indefinitely from the Language (BDEVIL) and his replacement was Barry Warsaw, who would be referred to as the Friendly Language Uncle For Life (FLUFL). The joke was codified in the next release.

More *Python* fun

```
>>> # Python uses != for "not equal" in comparisons
>>> 2 != 4
True

>>> # many other languages use <> for "not equal"
>>> 2 <> 4
  File "<stdin>", line 1
    2 <> 4
       ^
SyntaxError: invalid syntax

>>> from __future__ import barry_as_FLUFL
>>> # Barry was a proponent of using <> instead of !=
>>> 2 != 4
  File "<stdin>", line 1
    2 != 4
       ^
SyntaxError: with Barry as BDFL, use '<>' instead of '!='
>>> 2 <> 4
True
```

## 6 Three Important *Python* Object Types

Nearly everything in *Python* is an **object**. *Python* has three object types for values that are used very frequently; integers, floating point values (floats), and strings. Use `type(x)` to check the type of an object, i.e. `type(42)` or `type("Hello")`. Only integers and floats can be used to perform math operations.

- Integers
  - Whole numbers with no decimal part
  - **`int`** type
  - Values such as . . . , -3, -2, -1, 0, 1, 2, 3, . . .
- Floating point values
  - Non-integer numeric values and integers with a decimal point
  - Also known as floats
  - **`float`** type
  - Values such as 2.0, 2., and -3.45
- Strings
  - Any type of character or characters contained within quotes
  - **`str`** type
  - Double or single quotation symbols can be used. . .
    - `"Hello"`
    - `'Python'`
  - Numbers within quotes are strings
    - `"42"`
    - `'2.78'`

Determining object types

```
>>> type(3.14)
<class 'float'>
>>> type(42)
<class 'int'>
>>> type("Python")
<class 'str'>
```

### 6.1 Converting Types

Each of the three object types previously mentioned may be converted to one of the other types. Examples for each of the following will be provided in a code cell.

- To an integer. . .
  - Use the `int()` function
  - Convert floats - the decimal point and all numbers to the right are dropped
  - Convert strings that contain only an integer value
  - Cannot directly convert a string containing a float to an integer

- To a float...
  - Use the `float()` function
  - Convert integers - adds a decimal point and zero
  - Convert strings that contain only a float or an integer value
- To a string...
  - Use the `str()` function
  - Convert integers or floats - numeric values are simply enclosed in quotes

Type conversion examples

```
>>> int(3.14159)
3
>>> float(42)
42.0
>>> str(42)
'42'
>>> str(3.14159)
'3.14159'
>>> int("42")
42
>>> float("3.14159")
3.14159
>>> int(float("3.14159"))
3
```

## 7 Displaying Output Using `print()`

When working from a *Python* prompt or within a *Jupyter* notebook most results will be displayed implicitly (without the need to do anything special). However, within scripts the use of the `print()` function is required to explicitly display output. The `print()` function always provides more control of the displayed output (more on this at another time).

- Use the `print()` function to explicitly display results from a prompt or script
- The `print()` function displays the results of (nearly) anything included in the parentheses
- Multiple items/objects can be displayed with a single `print()` command
  - Items must be separated by commas within `print()`
  - A space is added between each item that was separated by a comma
- Multiple strings in a `print()` function can be connected together (concatenated) by separating them with the `+` symbol

Printing examples

```
>>> # Implicit "printing" from a Python prompt
>>> 2 + 2
4
>>> "Hello, World!"
'Hello, World!'

>>> # Explicit printing
>>> print("Python is awesome")
Python is awesome
>>> print("Hello " + "again")             # printing concatenated strings
Hello again
>>> print('Lumberjacks', "Parrots", 42)   # multiple items with commas
Lumberjacks Parrots 42
>>> print(4 * 5)                          # calculated then printed
20
>>> print(2 * 'Hello')                    # repeat strings by multiplying
HelloHello
```

## 8 Adding Comments in Your Code

It is a good idea and common practice to include comments in your code to describe "why" and not "how" something was done.

- Use the # symbol to indicate the start of a comment
- Anything after # is ignored (not executed) by *Python*
- Lines can start a with the # symbol; causing the entire line to be ignored
- Comments can be placed to the right of *Python* commands
- A common practice for *Python* scripts is to enclose multiple lines at the top of a script within triple quotes to form a single large comment (see below) called a docstring-style comment

Python comments

```
"""
This is a docstring-style comment
This program does such and such
It was written by Brian Brady
Last revision: 11/20/2022
"""

# This is a single-line comment
x = 42    # Answer to the universe and everything
print(x)
```

## 9 Basic Arithmetic

*Python* accepts common basic arithmetic operators; addition, subtraction, multiplication, division, and exponentiation.

- Addition: `2 + 2`
- Subtraction: `5 - 1`
- Multiplication: `2.5 * 4`
- Division: `7 / 3`
- Exponentiation: `4**2` (Note: the more commonly used `^` is used for a different operation)

*Python* adheres to the **PEMDAS** order of operations ...

1. **P** = Parentheses `( )`
2. **E** = Exponentiation: `4**2` is $4^2$
3. **MD** = multiplication and division from left to right: `3 * 2 / 3 = 2`
4. **AS** = addition and subtraction from left to right: `5 - 3 + 1 = 3`

Care should be taken when using multiple, different operators in an expression. Use parentheses where needed to group operations, but excessive overuse of parentheses can make expressions confusing and difficult to troubleshoot when there is an error.

Arithmetic operations

```
>>> 67 + 32 - 13
86
>>> 12 * 6 / 2
36.0
>>> 12 / 2 * 6  # same result as the previous calculation
36.0
>>> 42 - 10/4
39.5
>>> (42 - 10)/4  # parentheses make a difference
8.0
>>> 10 / 5 + 5
7.0
>>> 10 / (5 + 5)  # use parentheses when needed
1.0
>>> 2**3
8
>>> 9**1/2   # the 1/2 is not grouped, so 4**1 will be performed first
4.5
>>> 9**(1/2)
3.0
```

## 10 Special Division Operations

### 10.1 The Division Operation in Mathematics

Division involves four different objects: dividend, divisor, quotient, and remainder

- The dividend is divided up by the divisor
  - Dividend ÷ Divisor
  - Dividend / Divisor
- The quotient is the number of times the divisor completely goes into the dividend
- The remainder is what is left (if anything)
- For example, 7/2 results in a quotient of 3 and remainder of 1

### 10.2 Division in ***Python***

- Standard division, i.e `5 / 2`, always yields a floating point value
- Integer division, i.e. `5 // 2`, returns the quotient as an integer
  - Also referred to as floor division
  - Same as performing standard division and only keeping the part left of the decimal
- Modulo division, i.e. `5 % 2`, yields just the remainder
  - Also referred to as remainder division
  - If one value is divisible by another the remainder will be zero
- The built-in function `divmod(x, y)` requires two arguments (the dividend `x` and divisor `y`) and returns the quotient and remainder as a pair of values `(q, r)`

Division operators for 7/2

```
>>> print('Standard division: 7 / 2 =', 7 / 2)
Standard division: 7 / 2 = 3.5

>>> print('Integer division: 7 // 2 =', 7 // 2)
Integer division: 7 // 2 = 3

>>> print('Remainder (modulo) division: 7 % 2 =', 7 % 2)
Remainder (modulo) division: 7 % 2 = 1

>>> print('Quotient and remainder: divmod(7, 2) =', divmod(7, 2))
Quotient and remainder: divmod(7, 2) = (3, 1)
```

## 11 Elementary Math Functions and Constants

*Python* is a very extensible programming language. There are modules (libraries) available for nearly anything you can think of using. Common mathematical functions are provided in the `math` module.

- Must `import` modules whose commands you want to use before using the commands
- The `math` module includes math functions beyond basic arithmetic
- Use `dir(math)` or `print(dir(math))` after importing to see available `math` functions
- Must include the module name when calling functions, i.e. `math.log(2)`

Importing math and listing the available math commands

```
>>> import math
>>> print(dir(math))
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
↪  'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
↪  'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc',
↪  'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
↪  'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
↪  'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10',
↪  'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod',
↪  'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
↪  'trunc']
```

- Specific functions/commands can be imported from the `math` (or any other) module (see example below)
- Do not have to include `math.` for commands that are specifically imported

Importing specific math functions

```
>>> from math import cos, sin, tan, pi
>>> cos(pi/3)
0.5000000000000001
```

It is possible to import all functions from a module by using the `*` wildcard (as shown below). This is not considered good form nor is it very Pythonic and is frowned upon in most cases. Please do not use this method of importing modules unless specified by the instructor.

Not a good practice, but it works

```
>>> from math import *
>>> sin(pi/3)
0.8660254037844386
```

## 12 General Math Functions

- `abs(x)` $= |x|$ (absolute value is not part of the `math` module)
- `math.sqrt(x)` $= \sqrt{x}$ ($x$ must not be negative)
- `math.exp(x)` = `math.e**x` $= e^x$
- `math.log(x)` $= \log x$ or $\ln x$ (this is the natural log)
- `math.log(x, b)` $= \log_b x$ (the logrithm of base $b$)
- `math.log10(x)`$= \log_{10} x$
- `math.factorial(x)` $= x!$ ($x$ must be a positive integer)

General math functions

```
>>> abs(-100)
100
>>> math.sqrt(5)
2.23606797749979
>>> math.exp(2)
7.38905609893065
>>> math.log(math.exp(2))
2.0
>>> math.log(25, 5)
2.0
>>> math.log10(1000)
3.0
>>> math.factorial(5)
120
```

### 12.1 Math Constants

The following commands are used for the mathematical constants $\pi$, $e$ (Euler's number), $\infty$, and "not a number"

- `math.pi` $= \pi$
- `math.tau` $= 2\pi$
- `math.e` $= e$
- `math.inf` $= \infty$
- `math.nan` = "Not a number"

### 12.2 Trigonometric Functions

- The `math` module includes full support for trigonometric functions
- Standard trig functions require/use angle units of radians, not degrees
- Convert angles from degrees to radians using `math.radians(x)` (described later)

Trig functions

```
>>> math.sin(math.pi/4)
0.7071067811865475
>>> math.cos(math.pi/3)
0.5000000000000001
>>> math.tan(math.pi/4)
0.9999999999999999
```

- Inverse (arcus) trig functions return angles in radians, not degrees
- There is a second inverse tangent function named `atan2(y, x)`
  - Accepts two arguments `y, x` instead of one
  - Returns a quadrant-specific angle based on the $x$ and $y$ values
  - Arguments are in $y, x$ order to match $y/x$, i.e. `atan2(y, x)` equals $\tan^{-1}\left(\frac{y}{x}\right)$

Inverse trig functions

```
>>> math.asin(0.9)
1.1197695149986342
>>> math.acos(0.6)
0.9272952180016123
>>> math.atan(1.0)
0.7853981633974483
>>> math.atan2(-2, -4)
-2.677945044588987
```

- Angles can be converted from degrees to radians or from radians to degrees
  - `math.degrees()` converts an angle from radians to degrees
  - `math.degrees(math.pi/2)` converts $\pi/2$ to degrees
  - `math.radians()` converts an angle from degrees to radians
  - `math.radians(30)` converts $30°$ to radians

Trig functions with angle conversions

```
>>> math.sin(math.radians(30))
0.49999999999999994
>>> math.cos(math.radians(45))
0.7071067811865476
>>> math.degrees(math.atan(1.0))
45.0
>>> math.degrees(math.asin(3/5))
36.86989764584402
```

### 12.3 Rounding Related Functions

- `round(x, n)`
  - Rounds values towards zero if the decimal part is less than 0.5 and away from zero otherwise
  - Second argument is optional and is used to set the number of decimal places for rounding
  - `round(3.14159, 2)` will round to 2-decimal places, resulting in `3.14`
  - `round(8675309, -2)` will round to the hundreds place, resulting in `8675300`
- `math.trunc(x)`
  - Always drops off the decimal portion
  - Leaves an integer value
- `math.ceil(x)`
  - Returns the next integer value towards positive infinity if the argument has a decimal part
- `math.floor(x)`
  - Returns the next integer value towards negative infinity if there is a decimal part

Rounding related functions

```
>>> round(math.e)
3
>>> round(math.e, 3)
2.718
>>> round(8675309, -2)
8675300
>>> math.trunc(math.pi)
3
>>> int(math.pi)  # same as use math.trunc()
3
>>> math.ceil(2 * math.pi)
7
>>> math.ceil(2.0001)
3
>>> math.floor(44.99)
44
```

## 13 Using Names (Variables)

A great deal of the flexibility and power gained from using a programming language to automate tasks comes from the use of variables, also called names.

- Almost any value or object can be assigned to a name (variable)
- Variables can be used in place of the value or object after assignment
- Names in *Python* are case-sensitive, i.e. `a` and `A` are not the same

- Variable naming rules…
  - Must start with a letter or underscore
  - Can also contain numbers and underscores
  - No spaces and other special characters
  - Restricted *Python* names (keywords) cannot be used as variable names
  - Don't use built-in function or command names as variable names
    - For example, don't use `abs = 100` or `math.pi = 3`
    - Will overwrite the functions `abs()` and `math.pi`
- Using `del variable_name` will clear `variable_name`
- Try to use descriptive variable names like shown below

Restricted names (keywords) that cannot be used as variables

```
>>> help("keywords")

Here is a list of the Python keywords.  Enter any keyword to get more
 ↪  help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Assigning descriptive names (variables)

```
>>> answer_to_universe = 42
>>> print(answer_to_universe)
42
>>> radius = 10
>>> print('radius =', radius)
radius = 10
>>> bad_guy = "Darth Vader"
>>> total_time = 30
```

- *Python* does not display any results when a value or calculation is assigned to a name (see above). Use the `print()` function in order to explicitly display such results

## 14 Special Commands

- The last calculated result that was not assigned to a name may be accessed using the underscore `_` character
  - This is not normally done within a script or a function
  - Acts like the “ans” key on many calculators when working from a *Python* command line
- The `dir()` function returns a list of all current names and modules in memory
- `dir(module)` will return the available commands, functions, and methods in the module named `module`
- *Jupyter* and *iPython* have some *magic* commands
  - `who` is like `dir()` but formatted differently
  - `whos` is like `who` but it also includes types and value information
- Use `help()` to get interactive help
- Use `help(command)` to get help on `command`, i.e. `help(abs)`
  - Press the spacebar to go to the next page of a long help document
  - Press `q` to exit a document
- Use `help("topic")` to get help on `topic` instead of a command or function, for example `help("keywords")`